Extension du langage de requêtes LISQL pour la représentation et l'exploration d'expressions mathématiques en RDF

Sébastien Ferré

IRISA, Université de Rennes 1
Campus de Beaulieu, 35042 Rennes cedex, France
ferre@irisa.fr

Résumé : Les expressions mathématiques comptent pour une part importante dans les connaissances humaines. Nous en proposons une représentation en RDF afin de pouvoir les intégrer aux autres connaissances dans le Web sémantique. Nous étendons ensuite le langage de description et d'interrogation LISQL afin de concilier des représentations non-ambiguës, des requêtes expressives et des notations naturelles et concises. Par exemple, la requête `int(... ?X ^ 2..., ?X)` permet de trouver les intégrales en x dont le corps contient la sous-expression x^2. Tout cela permet d'utiliser Sewelis, un système d'information logique pour le Web sémantique, pour la représentation et l'exploration guidée d'expressions mathématiques. Ce guidage dispense les utilisateurs de maîtriser la syntaxe de LISQL et le vocabulaire tout en leur garantissant des expressions bien formées et des résultats à leurs requêtes.

Mots-clés : Web sémantique, RDF, expressions mathématiques, exploration.

1 Introduction

Les expressions mathématiques comptent pour une part importante dans les connaissances humaines et jouent un rôle central dans les documents scientifiques. Comme pour les autres formes d'information non-textuelle (ex., images, XML), il est donc important de pouvoir rechercher ces expressions par leur contenu (Youssef, 2006). Par exemple, on peut souhaiter retrouver les expressions contenant une intégrale en une certaine variable x contenant le terme x^2, telles que $\int x^2 + 1\, dx$ ou $\int y^2 - y\, dy$. Cet exemple fait apparaître deux difficultés de la recherche d'expressions mathématiques. La première est la nécessité de prendre en compte la structure

de l'expression, par exemple le fait que la sous-expression x^2 soit dans la portée d'une intégrale. La deuxième est la nécessité de s'abstraire du nom des variables liées, par exemple la variable x ci-dessus qui est liée par l'intégrale $\int dx$ et qui peut être renommée en y sans que cela ne change le sens de l'expression (α-équivalence). Les approches qui consistent à appliquer les méthodes de recherche dans les textes en linéarisant ces expressions (Miner & Munavalli, 2007) ne permettent pas de traiter ces difficultés (Youssef, 2006). Cela a pour conséquence de retourner des faux positifs tels que $\int 2x\,dx = x^2 + c$, et d'avoir des faux négatifs tels que $\int y^2 - y\,dy$. Les approches à base de langage de requête (Altamimi & Youssef, 2008; Kohlhase & Sucan, 2006; Guidi & Schena, 2003) permettent au contraire de bien traiter ces aspects en raisonnant directement sur la structure des expressions et en utilisant des "jokers" à la place de variables ou de sous-expressions. Dans le langage de Altamimi & Youssef (2008), la requête `\int ...$1^2... d$1` répond à la recherche ci-dessus.

Les Systèmes d'information logiques (LIS, Ferré & Ridoux (2004)) apportent un nouveau paradigme de recherche d'information combinant étroitement interrogation et navigation. Ils offrent ainsi aux utilisateurs l'expressivité de l'interrogation avec la facilité d'utilisation de la navigation. Initialement de puissance similaire à la recherche à facettes (Sacco & Tzitzikas, 2009), les LIS ont été récemment étendus aux données du Web sémantique avec comme résultat fort que la navigation offre une expressivité proche de celle de SPARQL et la garantie qu'aucun lien de navigation ne mène à un résultat vide (Ferré & Hermann, 2011). Le prototype Sewelis[1] guide les utilisateurs dans la construction incrémentale de requêtes arbitrairement complexes par ajouts successifs d'éléments pertinents. Cette construction incrémentale des requêtes et le souci de s'abstraire de notions de bas niveau (ex., algèbre relationnelle, logique) nécessite un langage différent de SPARQL : LISQL (LIS Query Language) (Ferré & Hermann, 2011). Par exemple, dans des données de type généalogiques, il est possible d'atteindre par navigation, c'est-à-dire sans rien saisir, la requête LISQL qui retourne les «personnes nées en 1601 ou 1649 quelquepart en Angleterre et dont le père est né à un endroit différent».

La contribution de ce papier est de montrer comment les expressions mathématiques peuvent être représentées en RDF (section 2) de telle sorte qu'il soit possible de faire des recherches prenant en compte leur structure et le renommage de variables liées (section 3). La syntaxe du langage LISQL doit cependant être enrichie pour que la présentation des expres-

[1] `http ://www.irisa.fr/LIS/softwares/sewelis`

sions et des requêtes se rapprochent de la notation mathématique (section 4). Nous détaillons ensuite un scénario complet de construction guidée de requêtes (section 5). À la section 6, nous complétons LISQL pour la représentation et l'exploration des structures de taille variable (ex., ensembles, vecteurs). Après une comparaison avec les travaux existants (section 7), nous terminons par une conclusion et des perspectives (section 8).

2 Représentation RDF et LISQL des expressions mathématiques

Un jeu de données RDF est un graphe orienté et étiqueté dont les noeuds sont des ressources (URI, *blank nodes* et littéraux) et dont les arcs $s \xrightarrow{p} o$ forment des triplets (s, p, o). Dans un triplet (s, p, o), p est une propriété RDF qui joue le rôle de prédicat et établit une relation entre le sujet s et l'objet o. Pour représenter les expressions mathématiques en RDF, nous profitons du fait que toute expression peut être représentée par un arbre, et donc par un graphe. Ces graphes permettent de plus de partager les différentes occurences d'une même variable ou d'un même opérateur.

Pour définir la représentation des expressions, nous nous inspirons des termes Prolog (Sterling & Shapiro, 1986) en raison de leur simplicité, de leur généricité et de leur adéquation avec RDF. En Prolog, un terme est soit un atome, soit un nombre, soit une variable, soit un terme composé. Les atomes sont des symboles logiques et correspondent bien aux URIs. Les nombres correspondent naturellement aux littéraux de RDF, lesquels peuvent également être des chaînes de caractères, des dates, etc. Comme en Prolog, les variables interviendront dans les requêtes mais pas dans les faits. Un terme composé est constitué d'un *foncteur* et d'une série d'arguments. Le foncteur est un atome et les arguments sont des termes quelconques. En supposant un espace de nommage `term:`, on peut définir les propriétés `term:functor`, `term:arg1`, `term:arg2`, ... pour lier la ressource représentant le terme composé à chacune de ses composantes. Afin de ne pas mélanger les notions d'identité et de contenu, on emploiera uniquement des *blank nodes* pour les termes composés et non des URIs. Il reste possible de lier une identité (URI) à un contenu (un terme) via une propriété *ad-hoc*, par exemple la propriété `rdf:value`.

La figure 1 donne la représentation graphique de la représentation RDF de l'expression $\int x^2 + 1\, dx$. Cette expression implique les opérateurs mathématiques `math:integral` (intégrale), `math:plus` (addition) et `math:power` (puissance), qui sont tous utilisés comme foncteurs d'arité 2. Dans le cas de `math:integral`, le second argument joue le rôle de va-

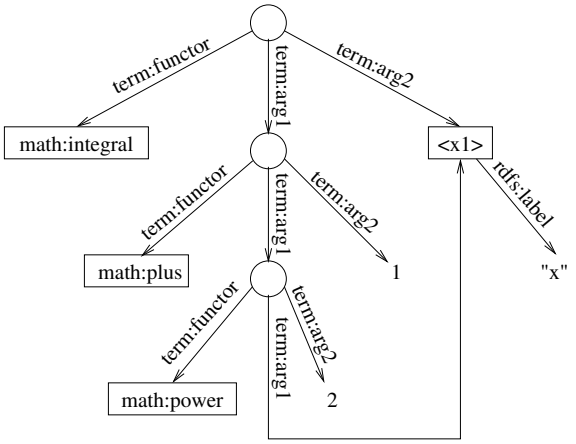

FIG. 1 – Représentation RDF de l'expression $\int x^2 + 1\, dx$.

riable liée. L'expression utilise également les littéraux entiers 1 et 2, ainsi qu'un URI local `<x1>` d'étiquette `"x"` pour représenter cette instance de la variable x. Cette représentation prend bien en compte tout le contenu de l'expression tout en s'abstrayant de certaines variations de présentation (ex., ajout de parenthèses, différentes notations pour l'intégrale). Elle permet également de distinguer des variables différentes bien que de même nom (ex., dans $x + \int x\, dx$) en utilisant différentes URI locales. Seule l'invariance au changement de nom de la variable liée n'est pas prise en compte dans la représentation de l'expression et sera traitée lors de la construction de requêtes.

Bien que les graphes RDF de formules puissent être sérialisés dans n'importe quelle notation RDF, nous utilisons ici LISQL qui est aussi lisible que la notation Turtle, tout en étant plus flexible, et d'expressivité similaire à SPARQL pour l'expression de requêtes. De plus, LISQL offre une syntaxe unifiée des descriptions et des requêtes. La notation du graphe de la figure 1 est la suivante.

```
term:functor : math:integral
term:arg1 :
    term:functor : math:plus
    term:arg1 :
        term:functor : math:power
        term:arg1 :
            <x1>
            rdfs:label : "x"
        term:arg2 : 2
    term:arg2 : 1
term:arg2 : <x1>
```

Comparé à Turtle, LISQL utilise l'indentation à la place des blocs (`[...]`) et sépare le prédicat et l'objet par un deux-point. Chaque niveau d'indentation correspond à une ressource. Le premier niveau correspond à l'expression entière, le deuxième correspond à la sous-expression $x^2 + 1$, le troisième à la sous-expression x^2 et le quatrième à la variable x. Une construction de la forme (p : o) au niveau d'une ressource s établit un triplet (s, p, o) où à la place de o on peut avoir un terme atomique (sur la ligne) ou un terme composé (sous-bloc en indentation).

3 Représentation LISQL des requêtes

Nous nous intéressons maintenant à la représentation de requêtes en LISQL pour la recherche d'expressions mathématiques. La table 1 donne les constructions syntaxiques du langage LISQL et leur traduction dans des *graph patterns* SPARQL. Une requête LISQL dénote un ensemble de ressources, ses réponses, et sa traduction $GP(x, q)$ prend donc en paramètre une variable x en plus de q. Le résultat d'une requête LISQL est l'ensemble des valeurs que peut prendre x pour vérifier $GP(x, q)$, soit le résultat de la requête SPARQL `SELECT ?`x `WHERE` $GP(x, q)$. Dans la syntaxe concrète de LISQL, le connecteur `and` est remplacé par des retours à la ligne et les parenthésages nécessaires par des indentations.

query q	*graph pattern* $GP(x, q)$
r	`FILTER (?`x `=` r`)`
`a` c	`?`x `a` c
p `:` q_1	`?`x p `?`y`.` $GP(y, q_1)$
p `of` q_1	`?`y p `?`x`.` $GP(y, q_1)$
`?`	`OPTIONAL { }`
q_1 `and` q_2	$GP(x, q_1)$ $GP(x, q_2)$
q_1 `or` q_2	`{` $GP(x, q_1)$ `}` `UNION {` $GP(x, q_2)$ `}`
`not` q_1	`FILTER NOT EXISTS {` $GP(x, q_1)$ `}`
`?`v	`FILTER (?`x `=?`v`)`

TAB. 1 – Les constructions syntaxiques du langage LISQL, et leur traduction en des *graph patterns* SPARQL : r est une ressource, c est une classe, p est une propriété, v est une variable LISQL, x et y sont des variables SPARQL (y est une variable fraîche).

Les variables LISQL jouent le même rôle que les variables en Prolog. Le point d'interrogation `?` est simplement une variable anonyme. Elles servent

de joker dans une requête en acceptant n'importe quel terme à leur emplacement. Elles servent également à poser une contrainte d'égalité entre deux atomes d'une expression. Cela permet de résoudre l'équivalence d'expressions au renommage des variables liées près : il suffit d'introduire une variable LISQL pour chaque variable liée. Par exemple, si on cherche les expressions $\int x\,dx$, $\int x^2\,dx$ ou $\int y^2\,dy$, on construira la requête suivante.

```
term:functor : math:integral
term:arg1 :
    ?X or
      term:functor : math:power
      term:arg1 : ?X
      term:arg2 : 2
term:arg2 : ?X
```

Pour traiter la requête de l'introduction où x^2 peut être un sous-terme du corps de l'intégrale, il reste à exprimer la relation de terme à sous-terme. Un triplet (s, `term:subterm`, o) représente le fait que le terme s contient o comme sous-terme à une profondeur quelconque. La relation de sous-terme est caractérisée par l'expression régulière $(arg1|arg2|\ldots)^*$ sur les chemins du graphe RDF. Pour donner cette sémantique à `term:subterm`, il suffit de déclarer dans l'ontologie des termes, exploitée par l'interpréteur LISQL, que les propriétés `term:arg1`, `term:arg2`, ... sont des sous-propriétés de `term:subterm`, et que la propriété `term:subterm` est transitive et réflexive. La recherche des intégrales en x contenant le terme x^2 peut alors se formuler comme suit.

```
term:functor : math:integral
term:arg1 :
    term:subterm :
        term:functor : math:power
        term:arg1 : ?X
        term:arg2 : 2
term:arg2 : ?X
```

Cette requête retourne les expressions $\int x^2\,dx$, $\int x^2+1\,dx$, $\int y^2-y\,dy$, mais pas les expressions $\int x^2+y\,dy$ et $\int 2x\,dx = x^2+c$. Maintenant, en partant du même exemple, supposons que l'on veuille obtenir le corps des intégrales plutôt que les intégrales elles-même. Il suffit pour cela de changer le point de départ de la requête et d'utiliser la construction p `of` q_1 de LISQL, qui permet de traverser les arcs du graphe RDF en sens inverse.

```
term:subterm :
    term:functor : math:power
    term:arg1 : ?X
    term:arg2 : 2
term:arg1 of
    term:functor : math:integral
    term:arg2 : ?X
```

Cette requête recherche un terme contenant le sous-terme x^2 et apparaissant comme le corps d'une intégrale en x.

4 Extension de LISQL pour des représentations plus compactes

f`(`a_1`,`a_2`,...)`	`term:functor :` f `term:arg1 :` a_1 `term:arg2 :` a_2 `...`

f`(this,`a_2`,...) :` q	`term:arg1 of`
	`term:functor :` f
	`term:arg2 :` a_2 `...`
	q

`...`q`...`	`term:subterm :` q
`...this... :` q	`term:subterm of` q

TAB. 2 – Nouvelles constructions syntaxiques de LISQL pour les termes et leur équivalence en LISQL originel. La deuxième construction se généralise pour n'importe quelle position de `this`.

Les exemples de la section précédente montrent que la syntaxe concrète de LISQL est lourde au point de rendre les descriptions et les requêtes presque illisibles et éloignées des notations mathématiques. La table 2 étend la syntaxe de LISQL avec de nouvelles constructions qui se réduisent simplement aux constructions existantes (sucre syntaxique). Là encore, nous nous inspirons de ce qui est fait en Prolog. La notation fonctionnelle $f(x, y)$ sert à représenter un terme composé. En remplaçant un argument par le mot-clé `this`, on forme une propriété complexe allant d'un terme argument à son terme parent : par exemple, `this^2` lie x à x^2, $(a+b)$ à $(a + b)^2$ et peut se lire "a pour carré". Ensuite, on représente les URIs (opérateurs et variables) par leur label, par exemple `+` au lieu de `math:plus` et `x` au lieu de `<x1>`. Enfin, pour rendre les représentations encore plus lisibles, les foncteurs binaires peuvent être déclarés comme opérateurs infixes avec une certaine priorité (ex., `x ^ 2 + 1` au lieu de `+(^(x,2),1)`), et les foncteurs unaires peuvent être déclarés comme opérateurs préfixes (ex., `- x ^ 2` au lieu de `-(^(x,2))`) ou postfixes (ex, `n !` au lieu de `!(n)`).

L'application de toutes ces améliorations aux exemples de la section précédente donne les représentations LISQL suivantes :

– `int(x ^ 2 + 1,x)` pour la description de $\int x^2 + 1\, dx$,

- `int( ?X or ?X ^ 2, ?X)` pour la recherche des intégrales en x dont le corps est x ou x^2,
- `int(... ?X ^ 2..., ?X)` pour la recherche des intégrales en x contenant x^2,
- `... ?X ^ 2... and int(this, ?X) : ?` pour la recherche des corps des intégrales en x contenant x^2.

5 Construction guidée de requêtes LISQL

Dans les sections précédentes, les requêtes sont données toutes faites, ce qui suggère que les utilisateurs doivent les saisir. Les problèmes liés à la saisie de requêtes sont les possibles erreurs de syntaxes (quelle est la grammaire du langage de requêtes ?), la méconnaissance du vocabulaire (quelles sont les URIs, classes et propriétés ?) et le manque de contrôle du volume des résultats (vide ou pléthorique). Une des motivations de la représentation d'expressions mathématiques en RDF est l'utilisation de Sewelis pour leur exploration. Cette section montre comment Sewelis permet de construire de façon guidée et incrémentale des requêtes complexes et pertinentes.

n.	notation math.	notation LISQL
1	$\int x^2\,dx$	`int(x ^ 2, x)`
2	$\int x^2+1\,dx$	`int(x ^ 2 + 1, x)`
3	$\int 2x\,dx = x^2+c$	`int(2 * x, x) = x ^ 2 + c`
4	$\int y^2-y\,dy$	`int(y ^ 2 - y, y)`

TAB. 3 – Un petit jeu de données formé de quatre expressions.

Nous utilisons ici un jeu d'expressions mathématiques assez simple pour que les relations entre le jeu de données et les réponses du système soient transparentes. La table 3 liste quatre expressions sous leurs notations mathématique et LISQL. Elles font intervenir les opérateurs arithmétiques usuels, l'égalité et l'intégrale.

Nous décrivons maintenant le scénario qui permet de construire la requête `int(... ?X ^ 2..., ?X)`. Après avoir chargé ce jeu de données dans Sewelis, la requête initiale par défaut est la requête vide `?` qui sélectionne toutes les ressources, c'est-à-dire les constantes, les opérateurs, les variables, les 13 termes composés mais également les classes et les propriétés connues. Parmi les suggestions d'incréments de requêtes, on trouve les schémas de termes composés associés aux opérateurs utilisés,

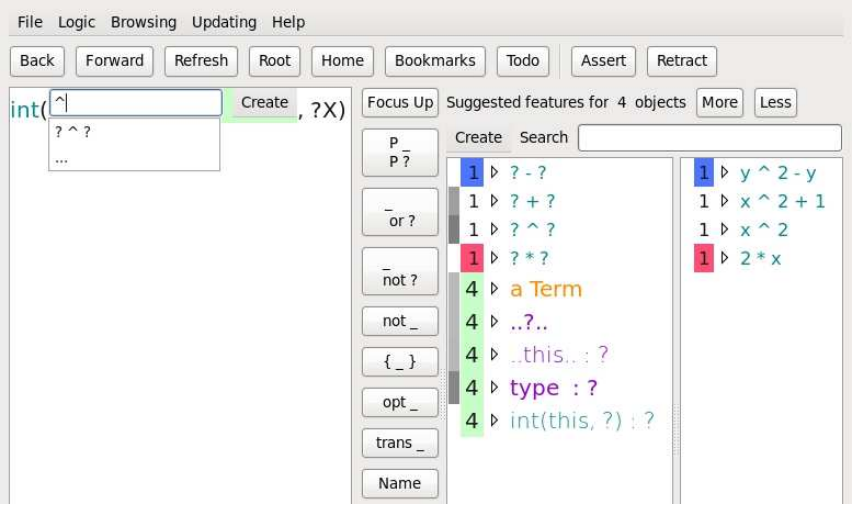

FIG. 2 – Une capture d'écran de Sewelis prise en cours de construction d'une requête. La requête est à gauche avec une zone de saisie matérialisant le focus. Les suggestions sont dans les deux colonnes à droite, colorées selon leur type (ex., termes en cyan, classes en orange).

par exemple `int(?,?)` ou `?+?`. Si on sélectionne le premier, la requête devient `int(?,?)`. Les réponses sont les 4 expressions et sous-expressions de type intégrale et la seule suggestion est `this=? :?` qui indique qu'une de ces intégrales apparaît comme terme gauche d'une égalité. Pour obtenir des suggestions sur une partie de la requête, on doit y positionner le *focus*, qui joue un rôle similaire au curseur d'un éditeur de texte. Toute sous-requête, et donc toute sous-expression, est un focus possible. Pour nommer la variable liée de l'intégrale, on déplace le focus sur le deuxième argument de l'intégrale. Les suggestions indiquent que la variable liée d'une intégrale est soit `x` soit `y`. On choisit d'insérer une variable LISQL `?X` (à l'aide du bouton *Name*) pour s'abstraire du nom de cette variable liée. On déplace ensuite le focus sur le premier argument de l'intégrale. Les suggestions nous indiquent qu'à cette position on peut trouver une opération parmi l'addition, la soustraction, la multiplication et la puissance (1 occurence chacun) et rien d'autre (voir figure 2). On a également la suggestion `...?...`, qui indique que toutes les expressions à cette position sont un sous-terme d'une autre expression, ce qui est valide pour tous les termes. On choisit cette dernière suggestion et on déplace le focus sur le `?` pour accéder aux sous-termes (au sens large) du corps de l'intégrale. La requête est

désormais `int(...?...,?X)`. Les suggestions sont les mêmes que précédemment plus la variable `?X` et le nombre d'occurence de `? ^ ?` est passé de 1 à 3 car 3 corps d'intégrales contiennent l'opérateur puissance. On sélectionne cette dernière suggestion pour contraindre une partie du corps de l'intégrale à être une puissance. Pour le deuxième argument de la puissance, la seule suggestion est 2, qu'on sélectionne. Pour le premier argument, les suggestions sont la variable LISQL `?X` et les noms de variables `x` et `y`. On sélectionne la variable LISQL pour faire le lien avec la variable liée de l'intégrale. En remettant le focus sur la requête entière, on obtient bien les résultats attendus, à savoir les expressions 1, 2 et 4 de la table 3.

On peut aboutir au même résultat en partant du terme `?X ^ 2` et en accédant ensuite aux termes le contenant en sélectionnant l'incrément `...this... :?`. Les suggestions indiquent que x^2 apparaît comme corps d'une intégrale, comme premier argument d'une addition et d'une soustraction, et également comme sous-terme d'autres termes. Après quelques étapes, on atteint la requête `?X ^ 2 and ...this... : int(this,?X) :?`, qui se lit comme «un terme de la forme x^2 qui apparaît comme sous-terme d'un terme qui apparaît lui-même comme corps d'une intégrale en x». En plaçant le focus sur le `?` final, on obtient les mêmes réponses que ci-dessus.

6 Représentation des ensembles, listes, vecteurs et matrices

On trouve en mathématiques des structures de taille variable et non-bornée, notamment les ensembles, les listes, les vecteurs et les matrices. A priori, on peut utiliser les termes pour représenter ces structures, en utilisant un foncteur pour chaque type de structure et un argument pour chaque élément de la structure. Par exemple, on pourrait représenter l'ensemble $\{1,2,3\}$ par le terme `set(1,2,3)` et le vecteur $(1\ 2\ 3)$ par le terme `vector(1,2,3)`. Un premier inconvénient est que, comme la taille de ces structures est non-bornée et potentiellement grande, cela suppose un ensemble non-borné de propriétés `term:arg1`, `term:arg2`... Un deuxième inconvénient est que cela ne permet pas de rechercher des structures contenant des éléments à des positions quelconques et dans un ordre donné, comme dans le langage de requête de Altamimi & Youssef (2008). Dans ce dernier, la requête `vector(...,a,...,b,...)` recherche les vecteurs contenant un a et un b dans cet ordre.

Nous réduisons le problème aux listes. En effet, une matrice est équivalente à un vecteur de vecteurs, un vecteur est isomorphe à une liste et pour

les ensembles, bien que l'ordre des éléments ne soit pas significatif, leur représentation concrète implique le choix d'un ordre. De plus, les listes font partie intégrante du standard RDF. Si on a la liste `[1,2,3]`, on peut en faire des ensembles ou vecteurs en lui appliquant différents foncteurs, `set` et `vector`, qui sont cette fois d'arité 1 quelque soit le nombre d'éléments.

Les listes peuvent être représentées par des combinaisons de termes. Dans de nombreux formalismes (LISP (Steele, 1990), Prolog (Sterling & Shapiro, 1986), RDF), une liste est soit la liste vide, soit une structure composée du premier élément de la liste et du reste de la liste, lequel est lui-même une liste. Cela implique deux foncteurs, c'est-à-dire deux constructeurs de listes. Le foncteur `term:nil` d'arité 0, qui correspond au symbole `[]` en Prolog, crée une liste vide. Le foncteur `term:cons` d'arité 2, qui correspond au symbole `(.)` en Prolog, crée une liste non-vide. Ainsi, le terme `cons(1,cons(2,cons(3,nil())))` représente la liste $[1, 2, 3]$. Ces deux foncteurs d'arité fixe et petite permettent donc de représenter des listes de longeur arbitraire.

En RDF, la liste vide est représentée par l'URI `rdf:nil` et une liste non-vide est représentée par un *blank node* qui est lié au premier élément par la propriété `rdf:first` et au reste de la liste par la propriété `rdf:rest`. La liste $[1, 2, 3]$ peut donc être représentée en LISQL comme suit.

```
rdf:first : 1
rdf:rest :
    rdf:first : 2
    rdf:rest :
        rdf:first : 3
        rdf:rest : rdf:nil
```

Par rapport à nos termes, le foncteur `term:cons` est implicite et les propriétés `term:arg1` et `term:arg2` sont remplacées par les propriétés `rdf:first` et `rdf:rest`. On peut donc facilement passer d'une représentation à l'autre et on les considérera comme équivalentes dans la suite.

Les exemples précédents montrent que la représentation à base de termes est un peu lourde comparé à la notation mathématique, et la notation LISQL encore plus. La notation Turtle[2] permet de représenter la liste $[1, 2, 3]$ sous la forme `(1 2 3)`, à la LISP. C'est satisfaisant, mais cela ne s'applique qu'aux descriptions de listes complètement instanciées. Pour représenter une requête telle que "quelles listes commencent par 1 suivi de 2", il faut revenir à la notation à base de termes, en l'occurence `cons(1,cons(2,?))`. Encore une fois, on s'inspire directement de Pro-

[2] `http://www.w3.org/TeamSubmission/turtle/`

log qui offre une syntaxe à la fois concise et couvrant aussi bien les descriptions que les requêtes. La table 4 donne la traduction des nouvelles constructions syntaxiques liées aux listes (les deux dernières constructions sont expliquées au paragraphe suivant). Celles-ci permettent de représenter la liste $[1, 2, 3]$ sous la forme `[1,2,3]`, donc de façon naturelle, et de représenter la requête retournant les listes commençant par 1 et 2 sous la forme `[1,2|?]`.

`[]`	`term:nil()` la liste vide
`[`e`|`l`]`	`term:cons(`e`,`l`)` une liste commençant par e dont le reste est l
`[this|`l_1`]` `:` l	`term:cons(this,`l_1`)` `:` l le 1er élément d'une liste l dont le reste est l_1
`[`e`|this]` `:` l	`term:cons(`e`,this)` `:` l le reste d'une liste l commençant par e
`[`e_1`,...,`e_n`|`l`]`	`[`e_1`|...[`e_n`|`l`]...]` une liste commençant par la série $e_1, \ldots, e_n$ et se terminant par la liste l
`[`e_1`,...,`e_n`]`	`[`e_1`,...,`e_n`|[]]` une liste composée des éléments $e_1, \ldots, e_n$
`[...|`l`]`	`term:sublist` `:` l une liste ayant pour sous-liste la liste l
`[...|this]` `:` l	`term:sublist of` l une sous-liste de la liste l

TAB. 4 – Nouvelles constructions syntaxiques pour les listes avec leur équivalence en LISQL et en français. Les symboles e (resp. l) dénotent des expressions LISQL quelconques représentant des éléments de listes (resp. des listes).

Il reste la question de pouvoir exprimer des sous-séquences dans les listes, par exemple "les listes contenant les éléments a et b, dans cet ordre". Comme les listes sont avant tout des termes, il est tentant d'utiliser la propriété `term:subterm` et de former la requête `...[a|...[b|?]...]...`. Cependant, les sous-termes d'une liste ne sont pas seulement ses éléments et ses sous-listes, mais également les sous-termes des éléments. La requête précédente aura donc des réponses inattendues, par exemple la liste `[z,a,z,[z,b,z],z]`. Nous introduisons donc une nouvelle propriété transitive `term:sublist` qui relie toute liste à chacune de ses

sous-listes. Cette propriété est donc la fermeture réflexive et transitive de la propriété `rdf:rest`. Les deux dernières lignes de la table 4 en donne une notation abrégée utilisant l'ellipse (...). En fait, cette ellipse peut être utilisée à la place de n'importe quel élément et dénote une suite d'éléments (éventuellement vide). La requête ci-dessus peut maintenant s'exprimer par `[...,a,...,b,...]`, qui est équivalente à `[...|[a|[...|[b|[...|[]]]]]]`.

7 Comparaison aux travaux existants

Nous comparons notre approche avec, d'une part, des langages de représentation d'expressions mathématiques, et d'autre part, des langages de requête pour la recherche d'expressions mathématiques (utilisés par exemple dans le cadre d'assistants de preuves, tels que Coq (Guidi & Schena, 2003)). À notre connaissance, aucune approche existante ne permet l'exploration guidée de telles expressions.

7.1 Langages de représentation

Le langage de référence pour la représentation d'expressions mathématiques est le langage MATHML[3], un dialecte de XML. En fait, MATHML définit deux langages : un langage de présentation et un langage de contenu. C'est ce dernier qui nous intéresse car il représente la structure logique d'une expression et évite les problèmes d'ambiguité (ex., e comme constante de Neper ou comme variable quelconque) et de synonymie (ex., x/y et $\frac{x}{y}$ pour la division). Le langage LaTeX joue le même rôle que le langage de présentation de MATHML, même s'il est parfois utilisé comme langage de contenu.

Le langage de contenu (strict) de MATHML est basé sur un petit nombre de balises XML qui encapsulent différents types d'expressions : `<cn>` (nombres), `<ci>` (identificateurs), `<csymbol>` (symboles), `<cs>` (chaînes), `<apply>` (applications), `<bind>` et `<bvar>` (quantifications). Par exemple, l'expression $\int x^2\,dx$ admet la représentation suivante en MATHML :

```
<bind><csymbol>integral</csymbol>
  <bvar><ci>x</ci></bvar>
  <apply><csymbol>power</csymbol>
     <ci>x</ci> <cn>2</cn>
  </apply></bind>
```

[3] `http ://www.w3.org/TR/MathML3/Overview.html`

Nous discutons ici de leur correspondance avec la représentation RDF. Les nombres et les chaînes sont naturellement représentés par des littéraux RDF de différents types (ex., `xsd:integer` pour les entiers, `xsd:string` pour les chaînes). Les symboles (ex., fonctions, opérations, constantes) sont naturellement représentés par des URIs, idéalement dans des vocabulaires standards. Les identificateurs peuvent aussi être représentés par des URIs, mais de préférence locaux au document les utilisant. Les applications (d'une fonction à des arguments) peuvent naturellement être représentées par nos termes. Enfin, les quantifications peuvent aussi être représentées par des termes en considérant le quantificateur (ex., $\exists$, $\forall$, $\int$) comme foncteur et en traitant les variables liées comme des arguments restreint à des identificateurs. La représentation LISQL de l'exemple précédent est ainsi `int(x ^ 2, x)`, où par convention, le deuxième argument de `int` désigne la variable liée.

Notre représentation RDF permet donc de représenter tous les contenus MATHML et ce de façon interopérable avec le langage de connaissance généraliste qu'est RDF. Par rapport à MATHML, cela permet de mixer librement connaissances mathématiques et connaissances non-mathématiques et également d'annoter librement les expressions et leurs constituants. Par exemple, le symbole `int` pourrait être décrit comme étant un quantificateur dont la variable liée apparaît comme second argument.

SPIN[4] utilise un vocabulaire semblable à nos termes pour la représentation des expressions de SPARQL en RDF. La notation OWL/RDF définit un vocabulaire *ad-hoc* pour la représentation RDF des axiomes OWL, qui sont des expressions. L'intérêt de notre approche est de définir un vocabulaire générique pouvant s'appliquer à tout type d'expressions : formules mathématiques, axiomes OWL, requêtes SPARQL ou même LISQL. Par exemple, la requête LISQL `?X^(1 or 2)` peut être représenté en LISQL par `lisql:term(^, lisql:var("X"), lisql:or(1,2))`, où le vocabulaire `lisql:` définit les foncteurs des constructions du langage LISQL. Appliqués à OWL, nos termes se rapprochent de la notation fonctionnelle de OWL (ex., `some(hasChild,Man)` en LISQL au lieu de `[a owl:Restriction; owl:onProperty ex:hasChild; owl:someValuesFrom ex:Man]` en Turtle), tout en restant compatible avec RDF. La syntaxe de Manchester[5] pour OWL peut ensuite être obtenue simplement en définissant les foncteurs du langage OWL comme des opérateurs infixes ou préfixes (ex., `hasChild some Man`).

[4] `http ://www.w3.org/Submission/2011/SUBM-spin-sparql-20110222/`

[5] `http ://www.co-ode.org/resources/reference/manchester_syntax/`

7.2 Langages de requêtes

Si on laisse de côté les approches à base de recherche textuelle dont les limites ont déjà été exposées dans l'introduction, on trouve des langages de requête qui opèrent directement sur la structure logique des expressions. MathWebSearch (Kohlhase & Sucan, 2006) définit un langage de requêtes XML qui étend MATHML. Sa syntaxe XML le rend très difficile d'utilisation et son expressivité est limitée. Par exemple, il ne permet pas d'exprimer la relation d'expression à sous-expression (ex., `...x ^ 2...`).

Le langage de requête le plus avancé est celui de Altamimi & Youssef (2008). Il utilise une notation ASCII aussi naturelle que peut l'être une telle notation et un jeu de 6 jokers qui peuvent être utilisés à la place de : un ou plusieurs caractères, un ou plusieurs atomes, une ou plusieurs expressions. Notre approche avec le langage LISQL offre une expressivité plus grande et un certain nombre d'avantages en plus pour l'utilisateur. LISQL offre une plus grande expressivité avec la disjonction, la négation et la possibilité de rechercher des sous-expressions apparaissant dans un certain contexte. Par exemple, il est possible de rechercher les corps d'intégrales en x contenant x^2 ou x^3 : `...?X ^ (2 or 3)... and int(this,?X) :?`. Les 6 types de jokers sont couverts par la combinaison de : la requête vide `?` (le joker universel), les variables LISQL pour les co-occurences d'un même atome, les ellipses `...` correspondant aux propriétés transitives `term:subterm` et `term:sublist`, et les requêtes LISQL classiques contraignant le type et le nom des différents atomes d'une expression. L'avantage principal de notre approche est que l'utilisateur n'a pas besoin de maîtriser la syntaxe du langage de requête puisqu'il est guidé pas à pas dans la construction des requêtes (et des représentations) avec la garantie d'obtenir des réponses. Un autre avantage est que des caractères UTF8 peuvent être utilisés dans les noms des symboles (ex., $\int$ au lieu de `int`), ce qui est possible du fait que ces symboles ne sont pas saisis par l'utilisateur, mais sélectionnés dans une liste de suggestions (ils sont également accessible par complétion automatique à partir du nom ASCII).

8 Conclusion et perspectives

Notre approche pour la représentation et l'exploration d'expressions mathématiques présente plusieurs qualités. Tout d'abord elle permet la représentation en RDF standard de ces expressions, assurant l'interopérabilité avec les autres connaissances représentées en RDF. Ensuite, le langage LISQL étendu combine à la fois des requêtes expressives et des notations

naturelles et concises sans lesquelles LISQL serait impraticable pour les utilisateurs finaux. Enfin, la navigation dans Sewelis permet de construire des expressions et des requêtes de façon entièrement guidée avec des garanties fortes sur leur cohérence avec la base existante. Ces résultats présentés pour des expressions mathématiques se généralisent à toutes données structurées pouvant se représenter sous forme de termes (ex., axiomes OWL, requêtes SPARQL).

Nous avons pour l'instant appliqué notre approche à un jeu de 70 expressions utilisant 34 foncteurs tirées du formulaire du bac S. Comme perspectives, nous envisageons de traiter des jeux de données plus larges, sans doute extraits de données MATHML, d'évaluer les performances de Sewelis et son utilisabilité.

Références

ALTAMIMI M. E. & YOUSSEF A. S. (2008). A math query language with an expanded set of wildcards. *Mathematics in Computer Science*, **2**(2), 305–331.

FERRÉ S. & HERMANN A. (2011). Semantic search : Reconciling expressive querying and exploratory search. In L. AROYO & C. WELTY, Eds., *Int. Semantic Web Conf.*, LNCS 7031, p. 177–192 : Springer.

FERRÉ S. & RIDOUX O. (2004). An introduction to logical information systems. *Information Processing & Management*, **40**(3), 383–419.

GUIDI F. & SCHENA I. (2003). A query language for a metadata framework about mathematical resources. In A. ASPERTI, B. BUCHBERGER & J. H. DAVENPORT, Eds., *Int. Conf. Mathematical Knowledge Management (MKM)*, LNCS 2594, p. 105–118 : Springer.

KOHLHASE M. & SUCAN I. (2006). A search engine for mathematical formulae. In J. CALMET, T. IDA & D. WANG, Eds., *Int. Conf. Artificial Intelligence and Symbolic Computation*, LNCS 4120, p. 241–253 : Springer.

MINER R. & MUNAVALLI R. (2007). An approach to mathematical search through query formulation and data normalization. In M. KAUERS, M. KERBER, R. MINER & W. WINDSTEIGER, Eds., *Calculemus/Mathematical Knowledge Management (MKM)*, LNCS 4573, p. 342–355 : Springer.

G. M. SACCO & Y. TZITZIKAS, Eds. (2009). *Dynamic taxonomies and faceted search*. The information retrieval series. Springer.

STEELE G. L. (1990). *Common Lisp : the Language*. Digital Press.

STERLING L. & SHAPIRO E. (1986). *The Art of Prolog*. Cambridge (MA) : MIT Press.

YOUSSEF A. (2006). Roles of math search in mathematics. In J. M. BORWEIN & W. M. FARMER, Eds., *Int. Conf. Mathematical Knowledge Management (MKM)*, LNCS 4108, p. 2–16 : Springer.